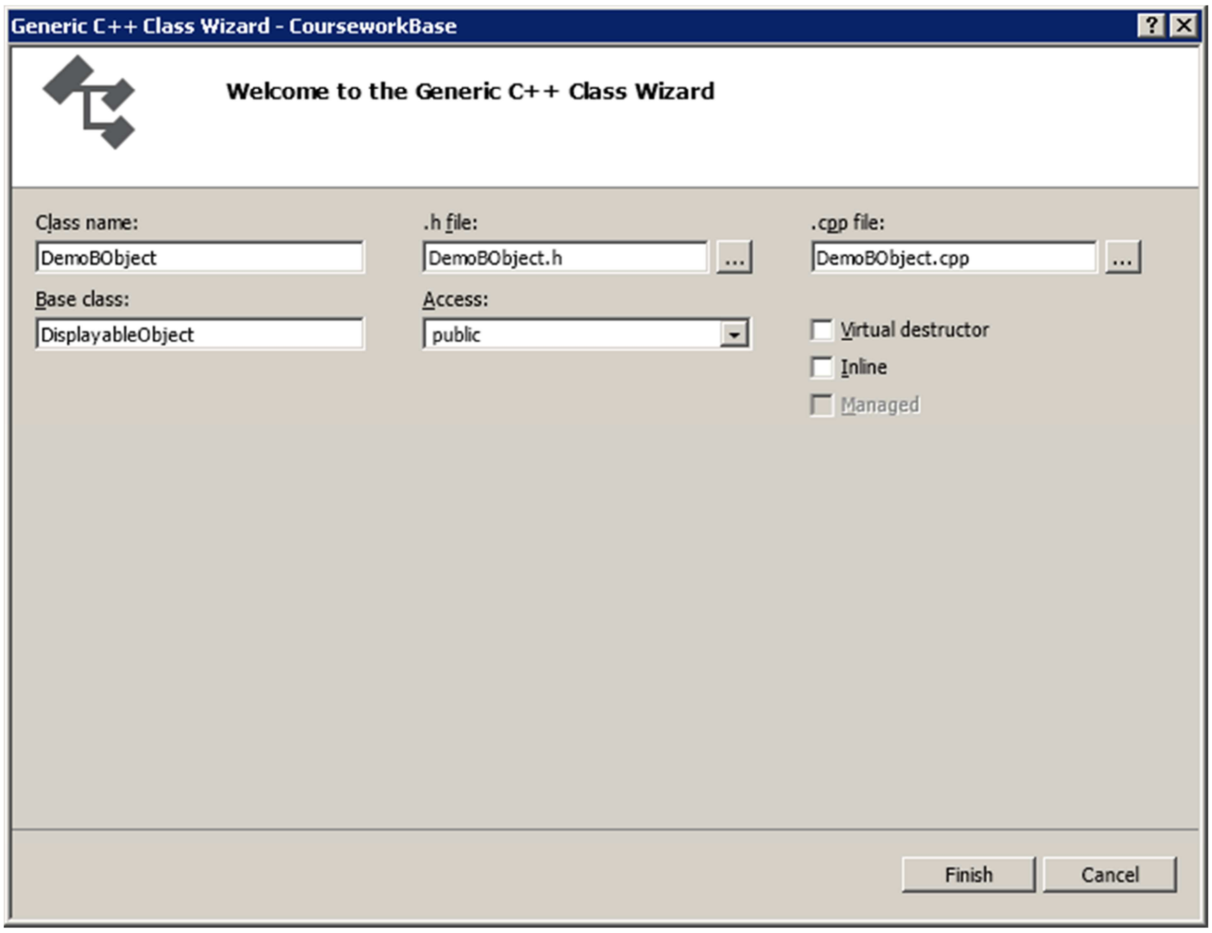
Coursework Lab B

It is extremely important that you finish lab A first, otherwise this lab session will probably not make sense to you. Lab B gives you a lot of the background and basics. The aim of the first of these two lab sessions was for you to learn the basics of how the coursework framework works. In the first session you say how to draw backgrounds and handle user input. In this second session you will learn about moving objects – drawing them and controlling them.

Load your files from the end of Lab A, since these will be your starting point.

Create a new class (right click on the Project name and choose 'Add' then 'Class', as in the previous lab session) called DemoBObject:


Generic C++ Class Wizard - CourseworkBase

Welcome to the Generic C++ Class Wizard

Class name: DemoBObject
.h file: DemoBObject.h
.cpp file: DemoBObject.cpp
Base class: DisplayableObject
Access: public
Virtual destructor: ☐
Inline: ☐
Managed: ☐

Here is the basic code which it will create:

```
class DemoBObject :
       public DisplayableObject
{
public:
       DemoBObject(void);
       ~DemoBObject(void);
};
```

There is a problem with the constructor which is created for you though, so if you try to build this it will not build. The base class constructor needs a pointer to the main program class. We need to accept this pointer in our own constructor. We will fix this now by changing the header file as follows:

```
class DemoBObject :
	public DisplayableObject
{
public:
	DemoBObject(BaseEngine* pEngine ); // THIS LINE CHANGED!!!
	~DemoBObject(void);
};
```

Now go to the .cpp file. You will now do the same thing as in the previous lab session:

1) Include the new header files at the top of the file:
   ```
   #include "header.h"
   #include "templates.h"
   ```

2) Change the constructor to take a single parameter, and to pass this to the base class constructor using the initialisation list (we will detail this in lectures):

   ```
   DemoBObject::DemoBObject(BaseEngine* pEngine )
   	: DisplayableObject( pEngine )
   {
   }
   ```

Your .cpp file should now be as follows:

```
#include "header.h"
#include "templates.h"

#include "DemoBObject.h"

DemoBObject::DemoBObject(BaseEngine* pEngine )
		: DisplayableObject( pEngine )
{
}


DemoBObject::~DemoBObject(void)
{
}
```

It should now compile correctly, but the object will not appear.

Modifying your moving object:

Your object will inherit a number of variables from its base class. You MUST set these variables appropriately because they are used to work out how to 'undraw' your object from the screen to perform animations.

Add the following code to the constructor:

```
DemoBObject::DemoBObject(BaseEngine* pEngine )
              : DisplayableObject( pEngine )
{
       // Current and previous coordinates for the object - set them the same
initially
       m_iCurrentScreenX = m_iPreviousScreenX = 100;
       m_iCurrentScreenY = m_iPreviousScreenY = 100;
       // The object coordinate will be the top left of the object
       m_iStartDrawPosX = 0;
       m_iStartDrawPosY = 0;
       // Record the ball size as both height and width
       m_iDrawWidth = 100;
       m_iDrawHeight = 50;
       // And make it visible
       SetVisible(true);
}
```

A position on the screen is stored. When the object is moved, the old position is stored in the PreviousScreen coordinates and the new position to draw at is stored in the CurrentScreen coordinates. For the moment, we set these to be the same thing, just to get an initial position.

The StartDrawPos variables specify where the start of the image (the top left corner) is, in relation to the position you tell the system to draw it on the screen. We will assume that our screen position is the top-left of the object so the offset is zero. When you finish this lab session, please consider and contrast the object in MyProjectMain.cpp with this one. The MyProjectMain.cpp object instead makes the screen position the centre of the object, so has an offset for the start draw positions.

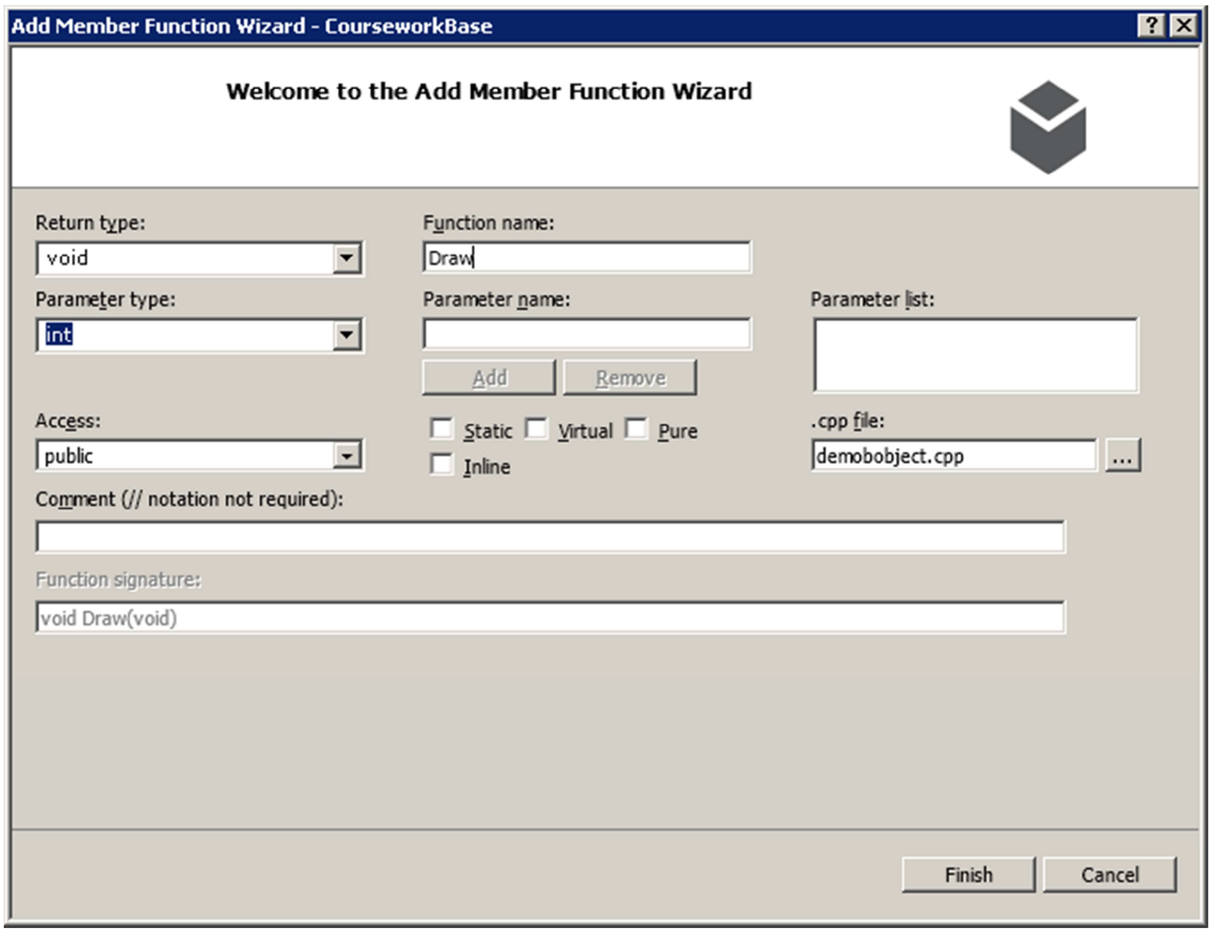
The width and height are hopefully obvious, and are in pixels.

Finally, the last function tells the object that it should be visible.

The next thing to do is to draw the object. We will just draw it as a solid rectangle for the moment.

Add a Draw() function

Add a function called Draw() with no parameters and void return type. (Right click on the class in Class View and choose “Add” and “Function”.)


Add Member Function Wizard - CourseworkBase

Welcome to the Add Member Function Wizard

Return type: void

Function name: Draw

Parameter type: int

Parameter name:

Parameter list:

Access: public

Static ☐ Virtual ☐ Pure ☐ Inline ☐

.cpp file: demobobject.cpp

Comment (// notation not required):

Function signature: void Draw(void)

Again this will add the function declaration to the header file and the implementation/definition to the .cpp file:

```
void DemoBObject::Draw(void)
{
}
```

Now add an implementation of the Draw() function as follows:

```
void DemoBObject::Draw(void)
{
	GetEngine()->DrawScreenRectangle(
		m_iCurrentScreenX, m_iCurrentScreenY,
		m_iCurrentScreenX + m_iDrawWidth -1,
		m_iCurrentScreenY + m_iDrawHeight -1,
		0x00ff00 );

	// This will store the position at which the object was drawn
	// so that the background can be drawn over the top.
	// This will then remove the object from the screen.
	StoreLastScreenPositionAndUpdateRect();
}
```

It is important to ensure that you only draw within the region that you specified. The drawing region is specified by:

The top left corner has the coordinates:

```
X:   m_iCurrentScreenX + m_iStartDrawPosX
Y:   m_iCurrentScreenY + m_iStartDrawPosY
```

The bottom right corner has the coordinates:

```
X:   m_iCurrentScreenX + m_iStartDrawPosX + m_iDrawWidth
Y:   m_iCurrentScreenY + m_iStartDrawPosY + m_iDrawHeight
```

In this case, we set the start draw position to 0,0 so the CurrentScreen values specify the top left corner. If you want to make the CurrentScreen values specify the middle of the object then instead you could set m_iStartDrawPosX to `-m_iDrawWidth` /2, and similarly for m_iStartDrawPosY. In other words, the StartDrawPos allows you to change the drawing of the object relative to the logical position of the object on the screen. If you don't need to do this then you can always set the StartDrawPos values to 0, but will have to remember that the position of the object is for the top left corner of the object.

Knowing this, the following code draws a rectangle filling the whole drawing area:

```
GetEngine()->DrawScreenRectangle(
	m_iCurrentScreenX, m_iCurrentScreenY,
	m_iCurrentScreenX + m_iDrawWidth -1,
	m_iCurrentScreenY + m_iDrawHeight -1,
	0x00ff00 );
```

- The -1 values are needed, because, for example, a rectangle of width 100, with the left side at position 0, would fill values from 0 to 99 (i.e. 100 pixels).
- The 0xff00 is a green colour, as for the background colours in lab A.
- DrawScreenRectangle means to draw the rectangle to the foreground. Screen means foreground, Background means draw to the background. Please review Coursework Lab A if you cannot remember the difference between the foreground and the background. Moving objects should be drawn to the foreground so that they can be 'undrawn' from their old positions when they move.
- GetEngine() retrieves a pointer to the BaseEngine object. In the constructor you took a pointer of this type and passed it to the base class constructor. The base class constructor stored that pointer for you. When you call GetEngine() you are retrieving the pointer which was stored. In this way you can call a function which is on the BaseEngine class (i.e. DrawScreenRectangle) even when you are in the DisplayableObject subclass.

If you build the project, it should build but you will still see nothing new on the screen.

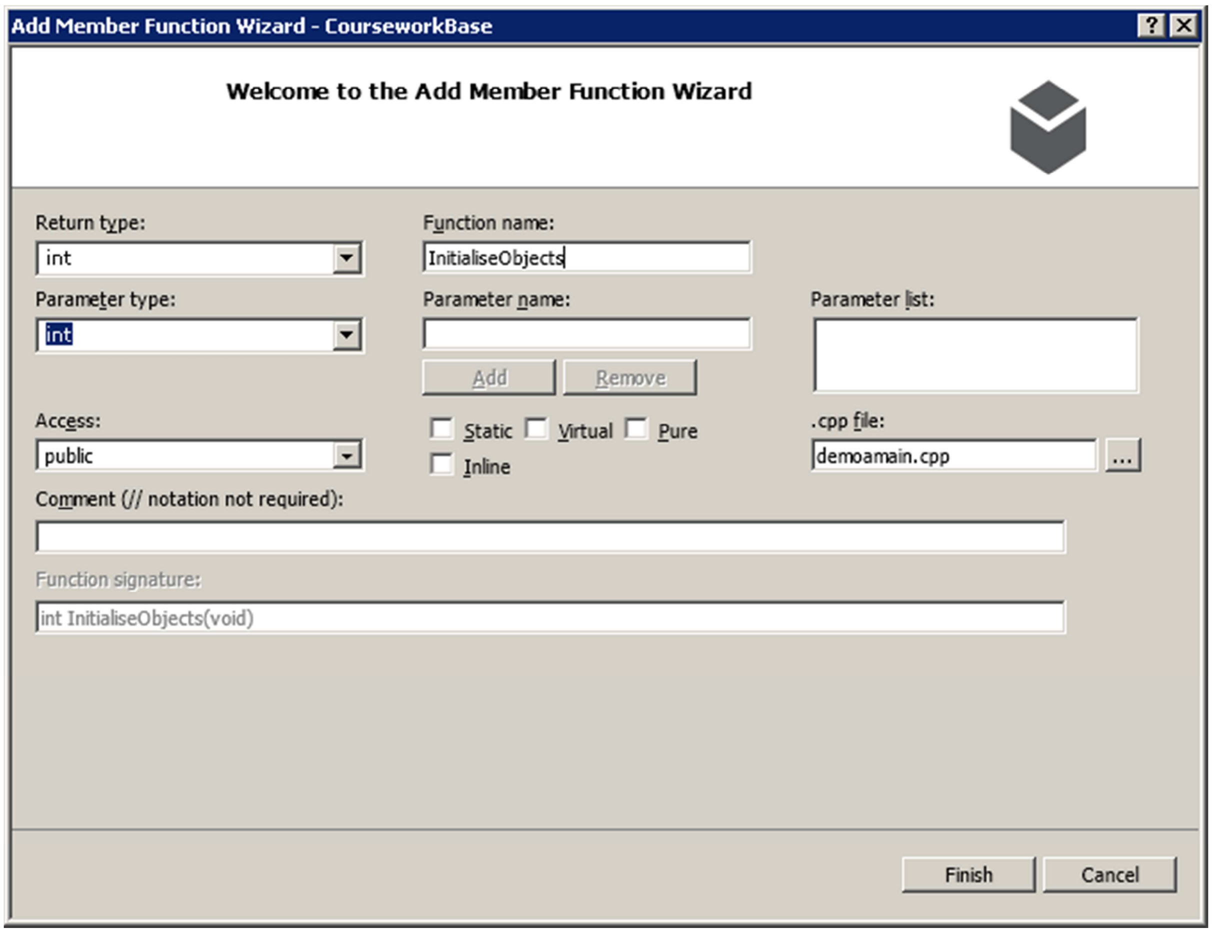
The `StoreLastScreenPositionAndUpdateRect()` function call to store the last position does a few things. It tells SDL that the object has been drawn, telling it to refresh the area within the region specified by:

```
X from        m_iCurrentScreenX + m_iStartDrawPosX
To            m_iCurrentScreenX + m_iStartDrawPosX + m_iDrawWidth
Y from        m_iCurrentScreenY + m_iStartDrawPosY
To            m_iCurrentScreenY + m_iStartDrawPosY + m_iDrawHeight
See the comments above about the drawing region for the object.
You can set these variables to whatever you want, but you must not draw outside of
this area. If you do then it may or may not appear on the screen, since
StoreLastScreenPositionAndUpdateRect will only tell SDL to draw this rectangle, not
the rest of the screen.
```

`StoreLastScreenPositionAndUpdateRect` will also store the current values of the CurrentPosition valiables in the PreviousPosition variables, so that the system knows where the object was drawn.

Create one of these objects in the BaseGame sub-class

Add a function to DemoAMain (NOT DemoBObject), called InitialiseObjects(), with an int return value and no parameters:


Add Member Function Wizard - CourseworkBase

Welcome to the Add Member Function Wizard

Return type: int

Function name: InitialiseObjects

Parameter type: int

Parameter name:

Parameter list:

Access: public

Static [ ] Virtual [ ] Pure [ ] Inline [ ]

.cpp file: demoamain.cpp

Comment (// notation not required):

Function signature: int InitialiseObjects(void)

Go to the DemoAMain.cpp file and find the new function:

```
int DemoAMain::InitialiseObjects(void)
{
	return 0;
}
```

This is a really important function. You need to create all of the objects which will be moving, and store pointers to them in the array. You will now add some code to it to create an object of type DemoBObject.

First, go to the top of the file, and add a #include for the header file, so that the top of the file looks like this:

```
#include "header.h"
#include "templates.h"

#include "DemoAMain.h"
#include "DemoBObject.h"
```

You need to add the include so that the compiler knows what a DemoBObject is when you use it.

Now add the following implementation to the InitialiseObjects function:

```
int DemoAMain::InitialiseObjects(void)
{
	// Record the fact that we are about to change the array - so it doesn't get
used elsewhere without reloading it
	DrawableObjectsChanged();

	// Destroy any existing objects
	DestroyOldObjects();

	// Create an array one element larger than the number of objects that you want.
	m_ppDisplayableObjects = new DisplayableObject*[2];

	// You MUST set the array entry after the last one that you create to NULL, so
that the system knows when to stop.
	// i.e. The LAST entry has to be NULL. The fact that it is NULL is used in
order to work out where the end of the array is.
	m_ppDisplayableObjects[0] = new DemoBObject(this);
	m_ppDisplayableObjects[1] = NULL;

	return 0;
}
```

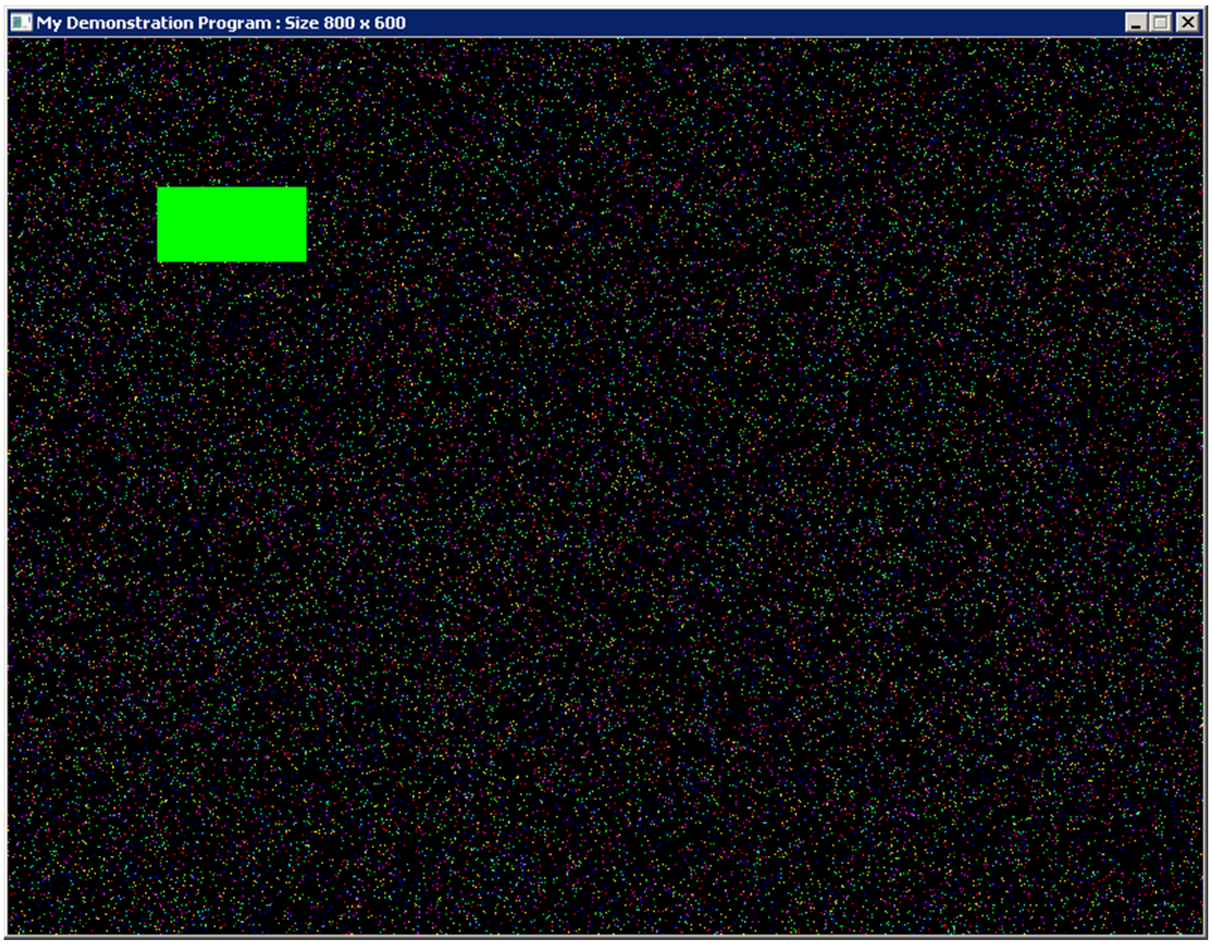
This does the following:

- Firstly it records that you have changed the drawable objects array. You MUST do this at the start of any implementation of this function.
- Next it looks at any objects which are stored in the array already and deletes them. This means that you never need to worry about destroying the objects, just call this function and it will do it for you.
- Now it creates a new array of DisplayableObject pointers. Note that this is an array of pointers. In this case the array has 2 elements. You must have one more element than you need displayable objects, because you must set the last element to NULL (much like for a C-type string).
- Finally, it sets the elements of the array to point to new objects of the correct type. In this case one DemoBObject, then a NULL to mark the end of the array.

If you wanted to create 3 objects you would just create an array of size 4, set the first three pointers to point to the new objects and then set element [3] to NULL.

Note that you MUST use new to create these objects since DestroyOldObjects() will use delete on them for you.

Build and test your program and it should look like this, showing the new object:

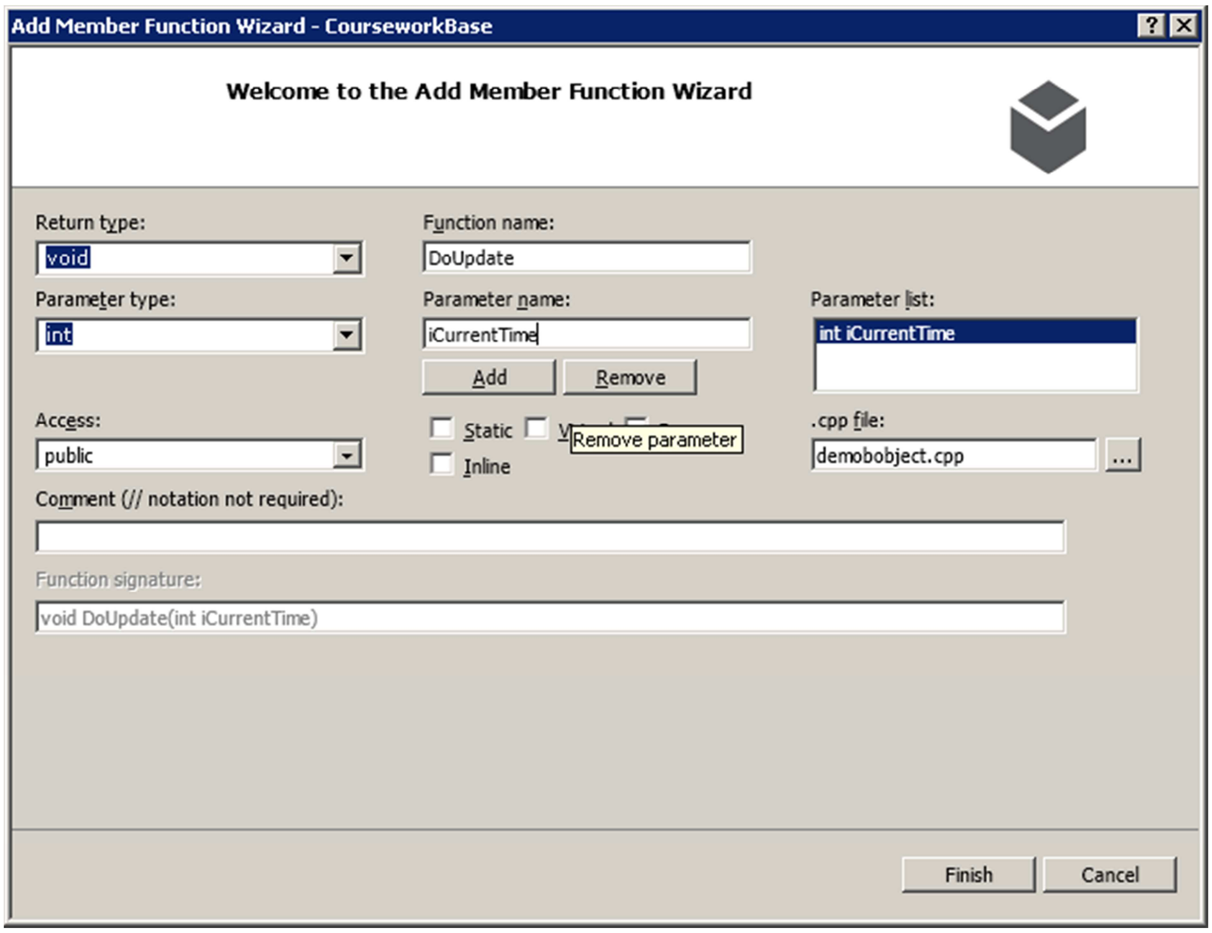


You can see the new object as the Green rectangle.

Making the object move

To move objects you need to implement their DoUpdate() methods.

Add a new function to DemoBObject called DoUpdate() with return type void and one parameter, of type int, with the name iCurrentTime.


Add Member Function Wizard - CourseworkBase

Welcome to the Add Member Function Wizard

Return type: void

Function name: DoUpdate

Parameter type: int

Parameter name: iCurrentTime

Parameter list: int iCurrentTime

Access: public

.cpp file: demobobject.cpp

Comment (// notation not required):

Function signature: void DoUpdate(int iCurrentTime)

The purpose of this function is to update the CurrentScreenX and CurrentScreenY variables, to change where the object will be drawn.

Add the following implementation for the function:

```
void DemoBObject::DoUpdate(int iCurrentTime)
{
	// Change speed if player presses a key
	if ( GetEngine()->IsKeyPressed( SDLK_UP ) )
		m_iCurrentScreenY --;
	if ( GetEngine()->IsKeyPressed( SDLK_DOWN ) )
		m_iCurrentScreenY ++;
	if ( GetEngine()->IsKeyPressed( SDLK_LEFT ) )
		m_iCurrentScreenX --;
	if ( GetEngine()->IsKeyPressed( SDLK_RIGHT ) )
		m_iCurrentScreenX ++;

	// Ensure that the object gets redrawn on the display, if something changed
	RedrawObjects();
}
```

Importantly, if you change any values you must call RedrawObjects at the end of this function. This will ensure that the Draw() function is called to draw the object in its new position. It will also ensure that the object is 'undrawn' from its old position, but this should just work for you as long as you set the member variables correctly.

Further information if you want it:

RedrawObjects() on the DisplayableObject class does the same as Redraw(false) on the BaseEngine (in fact it just calls Redraw(false)) and will tell the framework to redraw all of the moving objects. Each object will have recorded where it was drawn so that it can be 'undrawn'. The framework will go through its list of objects and draw the background back over where the object said it was drawn to. This will make the object disappear (i.e. the background will appear again because it has just been drawn again). Once all of the objects have been removed, the objects will all be asked to draw themselves again, in their new positions.

As long as there are not a lot of objects moving, just undrawing and redrawing the objects will be faster than redrawing the whole screen. If you have a lot of moving objects, you may just want to call `RedrawWholeScreen()` instead of `RedrawObjects()`, to just draw everything again. It may end up to be faster than the framework having to work out where everything is.

This example illustrates how you can check whether a key is currently pressed. You saw in the previous lab that you can implement a function which is called when a key is pressed. Instead this function will check not that that a key is being pressed down, but whether a key is already pressed down. For example, if a key is pressed twice then the OnKeyDown function would get called twice. On the other hand this function lets you check at any time whether the key is currently pressed down or not, not how many times it has been pressed.

These functions check whether the cursor keys are pressed. If a key is pressed then it changes the position of the object by 2 pixels.

Compile and test the program, trying to move the object using the cursor keys.

Further information if you want it:

As discussed in the last lab notes, GameAction() is called repeatedly by the framework, from the MainLoop() method. GameAction calls DoUpdate on each of the DisplayableObjects that it knows about. Your DisplayableObject inherits DoUpdate() from the base class, and since it is a virtual function, if you give your own implementation then your implementation will be called instead of the basic implementation (which does nothing).

At the moment you can move the object off the edge of the screen. You can add code to stop this quite easily, as show below:

```
void DemoBObject::DoUpdate(int iCurrentTime)
{
       // Change speed if player presses a key
       if ( GetEngine()->IsKeyPressed( SDLK_UP ) )
              m_iCurrentScreenY -= 2;
       if ( GetEngine()->IsKeyPressed( SDLK_DOWN ) )
              m_iCurrentScreenY += 2;
       if ( GetEngine()->IsKeyPressed( SDLK_LEFT ) )
              m_iCurrentScreenX -= 2;
       if ( GetEngine()->IsKeyPressed( SDLK_RIGHT ) )
              m_iCurrentScreenX += 2;

       if ( m_iCurrentScreenX < 0 )
              m_iCurrentScreenX = 0;
       if ( m_iCurrentScreenX >= GetEngine()->GetScreenWidth() - m_iDrawWidth )
              m_iCurrentScreenX = GetEngine()->GetScreenWidth() - m_iDrawWidth;
       if ( m_iCurrentScreenY < 0 )
              m_iCurrentScreenY = 0;
       if ( m_iCurrentScreenY >= GetEngine()->GetScreenHeight() - m_iDrawHeight)
              m_iCurrentScreenY = GetEngine()->GetScreenHeight() - m_iDrawHeight;

       // Ensure that the object gets redrawn on the display, if something changed
       RedrawObjects();
}
```

Again compile and execute this to test it.

That completes the tutorial on moving objects.

What to do now:

Please try the various demos.

I suggest that you start with the BouncingBall sample, then move on to Demo2, then Demo3 and finally Demo4. These give progressively more code to consider and more complex examples, building up to relatively complex programs. Using what you have learned you should be able to understand these.

However it is worth me explaining a few more things to help you to understand the various demos so I have added some additional information on the following pages.

Please also have a skim through the framework documentation. It may answer many of your questions:

http://www.cs.nott.ac.uk/~jaa/cpp/cw/SDL_framework_docs.pdf

Framework Overview

To give you an idea of how the framework fits together, and why implementing certain functions works, it is worth looking at the various function calls which are made in the framework.

Three functions are called from mainfunction.cpp:

- Initialise: This just sets up SDL, creates the window, etc.
- MainLoop: This is the interesting one and is discussed below.
- Deinitialise: This does the tidying up and frees resources.

MainLoop is the most important function for you. The table below shows the various functions which are called directly or indirectly by MainLoop. The functions in the first column are called directly by MainLoop. The second column shows the functions which these functions may call. The third column shows the functions which may be called by the functions in the second column, and so on. You will find that all of the functions which I showed you to override in these lab sessions are somewhere in this table, and you can see what calls them.

e.g. MainLoop will call KeyUp(), KeyDown(), MouseUp() and MouseDown() if it has received events, then it will always call GameAction() and GameRender(). GameRender() can call DrawScreen() if the screen needs to be redrawn, and that can call other functions, including DrawStrings(). DrawStrings() is discussed on the next page.

First function called by MainLoop:	**This calls…**	**This calls…**	**This calls…**
KeyUp			
KeyDown			
MouseUp			
MouseDown			
GameAction			
	UpdateAllObjects		
		DoUpdate *on every DisplayableObject*	
GameRender			
	DrawScreen		
		CopyAllBackgroundBuffer	
		DrawStrings	
		DrawChangingObjects	
	DrawChanges		
		UndrawChangingObjects	
		DrawStrings	
		DrawChangingObjects	
			Draw *on every DisplayableObject*

As mentioned before, please also have a skim through the framework documentation. It may answer many of your questions: **http://www.cs.nott.ac.uk/~jaa/cpp/cw/SDL_framework_docs.pdf**

Drawing strings which may change to the screen

If you just want to draw static strings then you can use DrawBackgroundString() to draw the string to the background within your SetupBackgroundBuffer() function. However, sometimes you want a string to change over time, such as when you are displaying a score or a timer.

Many of the demos have a DrawStrings function to do this, such as this from Demo3:

```
/* Draw text labels */
void Demo3Main::DrawStrings()
{
	CopyBackgroundPixels( 0/*X*/, 0/*Y*/, GetScreenWidth(), 30/*Height*/ );
	DrawScreenString( 150, 10, "Tile placement example", 0xffffff, NULL );
	SetNextUpdateRect( 0/*X*/, 0/*Y*/, GetScreenWidth(), 30/*Height*/ );
}
```

This can be used to draw text on the screen, if you don't want to have to model the strings as moving objects.

This is another virtual function and is called for your from GameRender(). To redraw the whole screen, GameRender() will call DrawScreen(), which will draw the background over the whole screen, then call DrawStrings(), then draw all of the moving objects. If you are just redrawing moving objects then GameRender() will call DrawChanges(), which will undraw the moving objects, call DrawStrings(), then draw all moving objects. In either case, DrawStrings will be called. This is why anything you draw in here will appear on the screen.

You need to do three things in this function:

1. Unless you have Redrawn the whole screen, you will need to remove the old strings from where they were drawn, otherwise you will just draw over them. You do this by copying the background image over the top of where the strings were drawn – undrawing them. CopyBackgroundPixels() is an easy way to do this, where you specify the top left coordinates of the rectangle to draw, and the width and height of the rectangle.
2. Next use the DrawScreenString function to draw a new string on the screen. If you do not specify a font then the default font will be used (see the start of coursework lab session A for how you specified this). Otherwise, to create a Font you should use the GetFont() function, as shown below.
3. Finally, you need to tell SDL that this part of the screen (where you drew the strings) has changed so that it will update it. The SetNextUpdateRect() function can be used for that.

Fonts

Important: all fonts which you wish to use need to be in the current directory when you run the program. If you run it from inside Visual Studio, this will probably be the src directory of your project. You need to put the .ttf file there if it is not already there.

```
void DemoAMain::DrawStrings()
{
	CopyBackgroundPixels( 0/*X*/, 0/*Y*/, GetScreenWidth(), 30/*Height*/ );
	DrawScreenString( 0, 10, "Tile placement example", 0xffffff,
GetFont("Cornerstone Regular.ttf", 50) );
	SetNextUpdateRect( 0/*X*/, 0/*Y*/, GetScreenWidth(), 30/*Height*/ );
}
```

If you need to load and use fonts, you can use the GetFont() function. There is a font manager in the framework. When you call GetFont() it will look for the font of the correct name, and create a font of the correct point size. The font will be loaded and kept in memory by the font manager. The next time you ask for that font and size it will just get the one which it loaded last time and reload it. This is important to know since it means that you are better off to keep reusing the same fonts/sizes rather than creating new ones. It also means that there MAY be a slight delay the first time you use a font (while it is loaded), but you should not notice it. Example of using GetFont():

Using a tile manager

A tile manager is a way to draw a grid of squares onto the background, and to update the screen whenever it is changed. You can use it as follows (see Demo4 and Demo4TileManager for an example):

First create a subclass of the TimeManager class.

Implement the GetTileWidth() and GetTileHeight() functions to specify the size of each tile, in pixels. (See Demo4TileManager.)

Implement DrawTileAt() to draw a tile. You can use GetValue() to retrieve the tile value to draw based upon the values of iMapX and iMapY. (See Demo4TileManager.)

Create the tile manager object at some point. E.g. add it as a member of your BaseEngine sub-class, so that it gets created when the BaseEngine sub-class is created.

At some point before using it, call SetSize() on the tile manager object. Also call SetBaseTilesPositionOnScreen() to specify the coordinates on the screen of the top-left corner of the visible tiles. See `Demo4Main::SetupBackgroundBuffer()`.

Call SetValue() for each tile to set the value of the tile. This value can then be used in the DrawTileAt() function to determine what tile to draw. See `Demo4Main::SetupBackgroundBuffer()`.

When you want to draw all of the tiles to the screen or background (e.g. to use them as a background image) call DrawAllTiles(). See `Demo4Main::SetupBackgroundBuffer()`. You have to specify which of the tiles you are drawing – i.e. how many rows and how many columns – so you can draw just a part of it if you wish.

If you want to change the value of a tile later on then you can call SetValue again on the tile manager.

If you want to change a tile value and have the tile manager immediately draw the new tile image to the screen (i.e. so that you see the change happen on the screen) then call UpdateTile() instead of SetValue(). This will bo the SetValue and then also tell the tile manager and SDL to redraw it. See `Demo4Object::DoUpdate( int iCurrentTime )` or `void BouncingBall1::DoUpdate( int iCurrentTime )`.

`void BouncingBall1::DoUpdate( int iCurrentTime ) is particularly interesting, especially the following code:`

```
	if ( m_pTileManager->IsValidTilePosition( m_iCurrentScreenX, m_iCurrentScreenY ) )
	{
		int iTileX = m_pTileManager->GetTileXForPositionOnScreen(m_iCurrentScreenX);
		int iTileY = m_pTileManager->GetTileYForPositionOnScreen(m_iCurrentScreenY);
		int iCurrentTile = m_pTileManager->GetValue( iTileX, iTileY );
		m_pTileManager->UpdateTile( GetEngine(), iTileX, iTileY, iCurrentTile+1 );
	}
```

This looks at the current position of the bouncing ball and asks the tile manager whether it is inside the grid. If so then it works out the map X and Y coordinates that the ball is over, then calls UpdateTile() too change the value of the tile. You should see the tiles change colour as the ball goes over them.

MovementPosition

This is a function to avoid you having to do any arithmetic yourself. The BouncingBall1 class illustrates how to use it:

Set up the movement:

```
m_oMovement.Setup( iStartX, iStartY, iEndX, iEndY, iStartTime, iEndTime );
```

This specifies a starting and ending position for the object, and a starting and ending time (in ticks). Functions such as DoUpdate() will have been passed the current time, so you can easily use this to decide a start and end time for the movement. E.g. from CurrentTime to CurrentTime+3000 (i.e. + 3 seconds).

Once this has been set up, you can ask it at any time to calculate what position it should be at that time by using the Calculate() function.

e.g.

```
        // Work out current position
        m_oMovement.Calculate(iCurrentTime);
        m_iCurrentScreenX = m_oMovement.GetX();
        m_iCurrentScreenY = m_oMovement.GetY();
```

If the movement has finished then you may want to make it do something else, e.g. to start the next move, or to move back again. You can easily do this by asking whether the movement has finished and if so call Setup() on it again, or just Reverse() to reverse the last move which was made, e.g.:

```
        // If movement has finished then request instructions
        if ( m_oMovement.HasMovementFinished( iCurrentTime ) )
        {
                m_oMovement.Reverse();
                m_oMovement.Calculate(iCurrentTime);
                m_iCurrentScreenX = m_oMovement.GetX();
                m_iCurrentScreenY = m_oMovement.GetY();
        }
```

The purpose of this class is basically to allow you to only have to worry about what will happen rather than having to calculate the positions of objects at any time. For more examples of use, please look at Demo3Object and Demo4Object.

What is the GameAction() function?

Many of the demos include an implementation of the GameAction() function. Basically this is the function which makes things change.

A basic implementation is here:

```
void MyProjectMain::GameAction()
{
	// If too early to act then do nothing
	if ( !TimeToAct() )
		return;

	// Don't act for another 10 ticks
	SetTimeToAct( 10 );

	// Tell all objects to update themselves
	UpdateAllObjects( GetTime() );
}
```

A tick in SDL is 1 millisecond. This code will ensure that UpdateAllObjects() gets called no more than once every 10 milliseconds at most. The TimeToAct() function asks whether it is time to act yet, and if not the function will end. The SetTimeToAct() function records that another action should not occur for 10ms after this one. The UpdateAllObjects() then gives all of the moving objects a change to move – i.e. that will call your DoUpdate() function on your objects.

If you wish to speed up or slow down the system, you can change the value 10, to change the delay between moves – speeding it up (adding lower delays) or slowing it down (adding bigger delays).

If you need to implement some facility which changes over time but is not associated with moving objects then you may want to add it to this function.

Managing a state model

This information should help you to understand parts of Demo4.

Managing a state model means having multiple states in your program. For example, an initial state, a playing state, a player loses state, a player won state, etc.

You then need to add some code to ensure that the screen is drawn differently in different states, and that different things happen in different states. Perhaps the simplest way to do this is:

- You need a variable to store the current state and you need to initialise it.
- When changing states, call InitialiseObjects() to create new objects and `SetupBackgroundBuffer()` to redraw the background. You will then also need to redraw the screen to get the updates to appear.
- Add a conditional statement (if or switch) to InitialiseObjects() to create different objects according to what needs to move in the current state. Remember that the old objects will get destroyed for you.
- Add a conditional statement (if or switch) to the SetupBackgroundBuffer() function to draw an appropriate background for the state.
- Add code to your functions which handle user input to ensure that they handle it in a way which is appropriate for the current state. E.g. if clicking on the screen should do nothing in the current state then don’t handle the mouse down in that state.